LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-PROC-666382

Performance Analysis and Optimization for BLAST, a High Order Finite Element Hydro Code

S. H. Langer, I. Karlin, V. A. Dobrev, M. L. Stowell, M. E. Kumbera

January 21, 2015

NECDC 2014
Los Alamos, NM, United States
October 20, 2014 through October 24, 2014

Disclaimer

(U) Performance Analysis and Optimization for BLAST, a High Order Finite Element Hydro Code

Steven Langer, Ian Karlin, Veselin Dobrev, Mark Stowell, and Mike Kumbera

langer1@llnl.gov, phone (925) 423-1358
Lawrence Livermore National Laboratory, Livermore, CA
Topical Areas: Hydrodynamics, Performance
Prefer: Talk

Abstract

BLAST [1, 2, 3] is an ALE (arbitrary Lagrange-Eulerian) hydrodynamics research code that supports arbitrarily high order finite elements. BLAST is being modified to add full multi-material support. High order elements with curved boundaries allow BLAST to compute correct answers on a much more distorted mesh than a normal second-order staggered grid hydro (SGH) code. The ability to run robustly with distorted meshes will provide significant advantages in usability.

This paper compares the performance of BLAST to Ares (a well-tuned SGH code) on an IBM Blue Gene/Q system. BLAST is run using several different orders for the finite elements. A large fraction of the run time of BLAST is spent in matrix multiplications and other linear algebra operators. Applying some relatively simple template based optimizations to small matrix operations and adjusting the order in which operations are performed improved the performance of BLAST by roughly a factor of 2.5. BLAST is now 2-4 or 6 times slower than Ares, depending on the mode in which it is run.

BLAST performs more floating point operations than Ares. The floating point performance of future computers is expected to increase faster than the memory bandwidth. This will bring the performance of BLAST closer to that of Ares without any further work. More complex optimizations will be studied in future work and should further close the performance gap. Our conclusion is that BLAST is an interesting candidate for the hydrodynamics package in a future multi-physics code.

Introduction

A computer chip with a fixed power budget and transistor count delivers the highest peak performance by having a large number of relatively slow cores. The "many core" approach to chip design should allow Moore′s Law improvements in performance per chip to continue for the next several years, but the bandwidth to off package memory will not grow fast enough to keep up with the increase in compute power. Some future computers will have fast in package memory, but it

will be too small to hold all the arrays needed by a typical multi-physics simulation.

Our current codes were designed in an era when off-package memory bandwidth was relatively high compared to compute performance. The performance of these codes on future systems is likely to track the aggregate memory bandwidth, not the aggregate computing power. We are investigating ways of better exploiting future computer systems.

BLAST [1, 2, 3] is an ALE hydrodynamics research code. It is built on top of the MFEM finite element library. BLAST started out as a Lagrangian code. Zone boundaries are curved in BLAST, so it can compute correct answers on a much more distorted mesh than a normal staggered grid hydro (SGH) code. However, the time step becomes quite small when the mesh becomes very distorted. For this reason an ALE capability was added to BLAST. BLAST runs more time steps between ALE remaps than a SGH code, which potentially reduces numerical diffusion. BLAST recently gained a multi-material Lagrange capability and support for multi-material ALE is under development.

This paper analyzes the performance of BLAST. The goal of the analysis is both to see how well BLAST performs today and to estimate how well it will perform on future computer systems. An IBM Blue Gene/Q system is used for the tests.

Table I: The computational intensity, GFLOP/s, and percentage of floating point peak are shown as a function of the element order for a Lagrangian run of the Sedov test problem. The tests were run on one node of BG/Q using a 2D 102,400 degree of freedom mesh.

	Q1Q0	Q2Q1	Q4Q3
% FLOPS	8.21	9.54	14.78
GFLOP/s	2.302	2.89	4.627
% of Peak	1.12	1.41	2.26

Using high order elements might allow us to obtain a solution of a given accuracy with a smaller memory footprint and fewer bytes transferred between memory and the CPU than a second order SGH code. This paper analyzes BLAST to determine how its performance, memory bandwidth, and other characteristics vary as a function of the finite element order. We do not yet have good accuracy metrics, so we compare the performance of BLAST to a traditional hydro code when both have the same number of degrees of freedom. This analysis will guide us to places where the current implementation can be improved.

An initial look at the properties of how BLAST changes with order is shown in Table I. This represents the baseline code in this paper. As shown in this table as the order increases the percentage of instructions that are floating point increases (%FLOPS). The GFLOP/s rate of the code and the percentage of machine peak achieved also increase as order increases. This result shows that switching to higher order codes does allow better utilization of the floating point capabilities of modern machines.

This paper is focused on the Lagrange algorithms in BLAST. There are many similarities in the Lagrange and ALE remap stages, so our results should apply to ALE simulations. In particular,

both phases solve mass matrices and spend most of their time performing matrix solves and dense linear algebra. A high order finite element diffusion package for BLAST is currently under development. When this package is working and coupled with BLAST, the cost of the advection step in the ALE remap will increase because the multi-group radiation field will have to be remapped along with the hydro variables. This paper does not analyze the diffusion package but, as with the ALE, data motion minimization and overhead reduction will be important to good runtime.

The focus of BLAST research to date has been on demonstrating the utility of high order finite elements and adding new features (e.g. ALE and multiple materials). The main branch of BLAST currently has too much overhead to perform well. In this paper we show that simple optimizations along with more complex algorithmic changes can significantly improve performance.

A performance comparison of Ares and BLAST

Ares is an unclassified LLNL radiation-hydrodynamics code. It has been under development for many years and is fairly well optimized. We use a 2D Sedov blast wave simulation to compare the performance of BLAST and Ares. The medium problem has 102,400 degrees of freedom (DOF) per node and the large problem has 409,600 DOF per node.

The order of the finite elements in BLAST is specified by two numbers via an expression like q1q0 or q4q3. The first number is for continuous basis functions (q1 gives second order accuracy) and the second number is for discontinuous basis functions (q0 gives second order accuracy). BLAST uses discontinuous finite elements for "zone centered" quantities (e.g. density and pressure) and continuous finite elements for "point centered" quantities (e.g. velocity and coordinates). BLAST (q1q0) has the same nominal accuracy as Ares.

Table II shows Ares is 5X faster than BLAST in SGH mode for the medium hydro test problem and 8X faster for the large hydro test problem. We need to determine whether the lower performance of BLAST is fundamental or whether it is due to the current implementation of BLAST. BLAST moves less data to and from memory than Ares, but executes 16.3X/ 24.5X times as many integer instructions and 2.6X times as many floating point instructions when running with q1q0 elements. BLAST is noticeably more floating point intensive than Ares, but moves somewhat less data.

Given that current machines can perform multiple floating point instructions in the time it takes to move a byte from main memory, the extra floating point instruction executed by BLAST are not a fundamental limitation. However, for higher order computations, BLAST moves significantly more data from memory than Ares as well as performing more floating point computations. Therefore, the current implementation is likely to always be slower than Ares.

The reason for this gap is that the higher order versions of BLAST must solve a mass matrix that involves a significant number of iterations. When this solve is added to the q1q0 code (non SGH mode) its data motion increased by a factor of six or more. Therefore, both the low order and high order implementations of BLAST that use the current linear solver will always be slower than a code such as Ares per degree of freedom. However, there are other ways to solve linear systems as we detail next.

Memory Usage

Table II: Ares and BLAST were used to run medium and large sized 2D test problems. BLAST was run with both q1q0 and q4q3 to demonstrate the effects of using higher order elements. The default version of BLAST issues significantly more integer and floating point instructions than Ares.

Code	Problem	Bytes moved to and from memory (1e9)	Integer instructions (1e9)	FP instructions (1e9)	Runtime (seconds)
Ares	medium	36.76	17.26	8.03	2.31
BLAST q1q0 sgh	medium	23.09	281.60	20.71	17.02
BLAST q1q0	medium	142.83	453.28	38.65	40.76
BLAST q2q1	medium	303.43	462.45	48.77	32.80
BLAST q4q3 rk4	medium	965.13	2055.04	274.16	129.50
Ares	large	159.09	45.88	31.54	8.48
BLAST q1q0 sgh	large	129.39	1126.69	82.75	66.56
BLAST q1q0	large	1391.05	1706.39	151.75	131.50
BLAST q2q1	large	1451.95	1792.16	192.02	131.04
BLAST q4q3 rk4	large	4031.35	8164.53	1085.61	509.83

One of the most important factors determining the performance of hydro codes and multi-physics codes is the amount of data transferred between the processor caches and external memory. The amount of data transferred depends on the amount of memory in use and the number of times each datum must be transferred between memory and the CPU in the course of a time step. As mentioned in the previous section, the solve phase of BLAST contains most of its data motion.

Fig. 1 shows the total number of MB of memory in use by BLAST as a function of the order of the finite elements. The leftmost point is the memory used by BLAST in SGH mode. The second order BLAST run (q1q0) uses only slightly more memory than the SGH mode. The memory use by BLAST increases rapidly in going to 4th order elements. This is primarily due to the memory used in the matrix solve. Partial assembly (see below) is an alternate approach to the mass matrix solve that has flat memory usage as order increases. It also requires significantly less data motion than the fully assymbled code shown above.

Partial Assembly

BLAST spends most of its time on two tasks. The first is computing corner forces. Corner forces are evaluated once per mass matrix solve and don′t require a global solve. Computing corner forces requires evaluating a stress tensor at each quadrature point. This may be expensive for some material models. The second task is the mass matrix solve. The mass matrix has global coupling and must be solved iteratively. A fairly typical value is 30 iterations.

Corner forces are compute bound and take most of the time with a few threads. The solve phase is memory bound for more than 16 threads on a Blue Gene/Q and takes almost as long as corner forces when all 64 threads are used. Figure 2 shows this clearly as the mass matrix solve only scales well up to 16 threads where it starts to become memory bound, while corner forces scales out to all threads. The mass matrix solve will become the dominant portion of the runtime on future

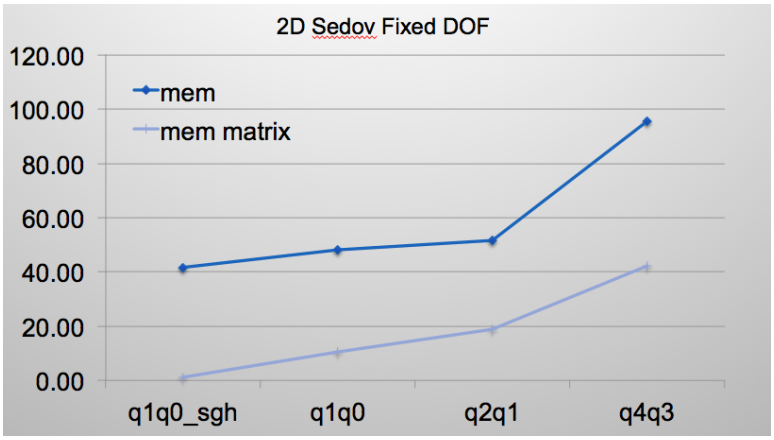
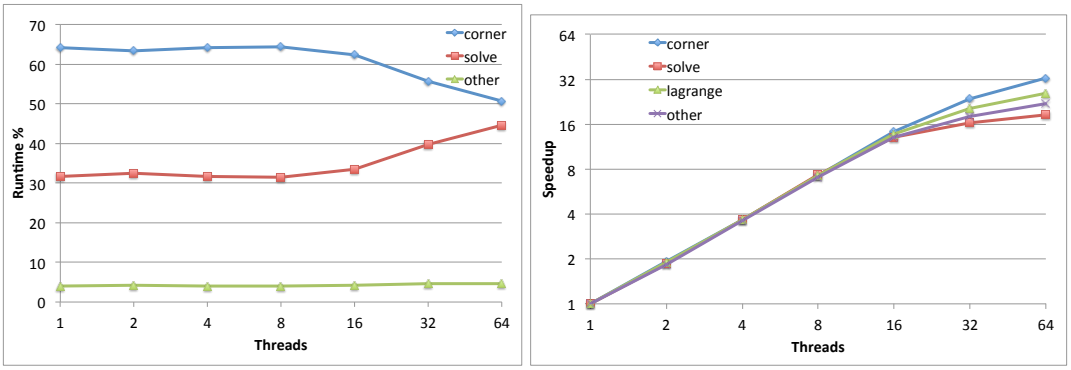


Figure 1: The total amount of memory used increases with finite element order with the number of degrees of freedom held constant. Most of the increase is due to memory used to store matrices.



(a) Runtime Percentage (b) Scaling

Figure 2: Runtime percentage and scaling of 102,400 Degree of Freedom 2D Sedov problem on BG/Q.

machines if, as expected, compute capacity continues to grow faster than memory bandwidth.

There are many ways of solving the mass matrix equation. We now consider an approach that is faster than the default implementation. The mass matrix equation is solved iteratively for the solution vector x.

$$Ax = b$$

$$A = P^T G^T B^T D B G P$$

The A matrix is the product of several smaller matrices. B and D are small dense matrices. P and G are global sparse matrices.

Certain matrix products (e.g. BGP) don't change between Lagrangian time steps. The default method in BLAST multiplies the matrices once and stores the results to reduce FLOP counts. The problem with this approach is that the resulting sparse matrices are large. Partial assembly keeps

the matrices separate and applies them sequentially to the vector on every iteration. This has a smaller memory footprint, but requires more operations for low orders.

Table III: The base partial assembly version usually performs better than the default BLAST implementation for the medium size test problem. The tuned partial assembly version outperforms the default version for all tests.

Method function	Default corner (sec)	Default solve (sec)	Partial corner (sec)	Partial solve (sec)	Tuned partial corner (sec)	Tuned partial solve (sec)
Q2Q1 2D	16.6	14.6	15.8	22.6	6.5	8.4
Q2Q1 3D	34.9	25.3	24.9	23.8	11.9	8.9
Q4Q3 RK4 2D	64.6	47.7	29.2	30.4	13.8	18.2
Q4Q3 RK4 3D	696.2	95.1	48.6	37.1	22.9	23.0

Table III shows that the base partial assembly version is faster than the default version for all but the Q2Q1 2D mass matrix solve. Performance counters show that the base partial assembly version moves less data than the default version but has 3x the integer instructions and over 2x the floating point instructions in its preconditioned conjugate gradient solve. These differences result in a code that issues over 0.8 integer instructions per cycle when the processor has a maximum throughput of 1. Therefore, BLAST went from memory bandwidth bound to integer instruction bound by switching to partial assembly. In 2D, where less data is moved per iteration in the fully assembled code, this tradeoff resulted in worse performance.

That does not mean we need to abandon partial assembly. The quadratures in BLAST are carried out using multiplication of small, dense matrices. Some matrices are always 2x2 or 3x3 depending on the mesh dimensionality. The size of others increases as the order of the elements increases. The baseline version of BLAST has two linear algebra options. It can use generic MFEM functions for all dense linear algebra or optimized BLAS and LAPACK calls. The generic MFEM functions are typically used except for very high orders due to the overhead of selecting the right BLAS function and because BLAS is typically tuned for large matrices. The generic MFEM functions still incur significant function call and loop iteration overhead. These overheads are integer instructions and can be significant for small matrices.

Earlier work demonstrated that inlining functions and unrolling loops by hand for small matrices improves performance. However, code written in that format can be hard to maintain and modify. By using C++ templates to inline and specialize functions for their exact size we are able to create a code that is easier to maintain, read, and specialize for arbitrary dimensions and orders. The last two columns in Table III show the results of specialization using templates. Speedups of over 2x are seen for most orders shown. It is also important to note that for lower orders and 2D (where more function calls and smaller loops are used) the template specialization has a large impact.

The version of BLAST using templates executes a similar number of integer instructions in its mass matrix solve as the original code, but runs significantly faster because it is not memory bandwidth bound. The reductions occurred both because of removing unneeded operations when dimensions were known at compile time and due to the compiler performing extra optimizations, such as loop unrolling, because it knew at compile time the extent of the loops.

Future Work

Taking advantage of interior degrees of freedom

Good spatial coherence between memory accesses is a well known way of obtaining good cache utilization. For high order finite elements, degrees of freedom in the interior of zones can be assigned to contiguous blocks of memory. This simple re-ordering of data storage should allow high order BLAST runs to achieve cache utilization similar to a structured grid code while using an unstructured mesh.

SIMD Operations

The computational intensity of BLAST is great enough at high orders that it should be able to achieve speedups through the use of single-instruction-multiple-data (SIMD) operations. Our first step will be to hand optimize selected kernels so that SIMD operations are used and see if this leads to the expected speedup. Our longer term goal is to develop an abstraction that allows BLAST to take advantage of SIMD instructions without making the source code more complicated. A significant amount of the run time is spent in the matrix operations discussed in the previous section. A fairly high fraction of SIMD instructions could be issued by using matrix libraries that have been SIMD-ized.

Accuracy

BLAST and Ares are both first order accurate at shocks. BLAST can deliver a more accurate answer for a given number of DOF than Ares away from shocks. A topic for future work is determining the relative performance of Ares and BLAST when they deliver the same global accuracy.

Conclusions

High order finite element hydro has a computational intensity better suited to future hardware than SGH. Ares was initially 5-10X faster than the standard implementation of BLAST in SGH mode. However, with partially assembled operators and optimizations for small dense linear algebra, the ratio drops to 2-4X. The ratio for full finite element mode is roughly 6X. The optimizations we investigated improve performance without making writing and maintaining BLAST significantly more difficult. Additional work is required to further close the performance gaps and quantify the accuracy gains from high order methods, but the usability advantages of BLAST are enough that even with the current performance it is an interesting possibility for use as a hydro package in a future multi-physics code.

Acknowledgements

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC. Document releases LLNL-ABS-655720, LLNL-PRES-661954.

References

[1] Dobrev, V., and Kolev, Tz., and Rieben, R. "High-Order Curvilinear Finite Element Methods for Lagrangian Hydrodynamics," *SIAM Journal on Scientific Computing*, **34**:B606-641 (2012).

[2] Anderson, R., and Dobrev, V., and Kolev, Tz., and Rieben, R. "Monotonicity in High-Order Curvilinear Finite Element ALE Remap," *International Journal for Numerical Methods in Fluids*, **10**:1002 (2014).

[3] BLAST home page, www.llnl.gov/casc/blast.